8

Productivity Engineering in the UNIX† Environment

A Shared Object Hierarchy

Technical Report

S. L. Graham
Principal Investigator

(415) 642-2059

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Contract No. N00039-84-C-0089

August 7, 1984 - August 6, 1987

Arpa Order No. 4871

†UNIX is a trademark of AT&T Bell Laboratories

A Shared Object Hierarchy[†]

Lawrence A. Rowe

Computer Science Division, EECS Department
University of California
Berkeley, CA 94720

Presented at the
International Workshop on Object-Oriented Database Systems

September 23-26, 1986, Pacific Grove, California

Abstract

This paper describes the design and proposed implementation of a shared object hierarchy. The object hierarchy is stored in a relational database and objects referenced by an application program are cached in the program's address space. The paper describes the database representation for the object hierarchy and the use of POSTGRES, a next-generation relational database management system, to implement object referencing efficiently. The shared object hierarchy system will be used to implement Object FADS, an object-oriented programming environment for interactive database applications that will be the main programming interface to POSTGRES.

1. Introduction

Object-oriented programming has received much attention recently as a new way to develop and structure programs [GoR83,StB86]. This new programming paradigm, when coupled with a sophisticated interactive programming environment executing on a workstation with a bit-mapped display and mouse, improves programmer productivity and the quality of programs they produce.

A program written in an object-oriented language is composed of a collection of objects that contain data and procedures. These objects are organized into an *object hierarchy*. Previous implementations of object-oriented languages have required each user to have his or her own private object hierarchy. In other words, the object hierarchy is not shared. Moreover, the object hierarchy is usually restricted to main memory. The LOOM system stored object hierarchies in secondary memory [KaK83], but it did not allow object sharing. These restrictions limit the applications to which this new programming technology can be applied.

There are two approaches to building a shared object hierarchy capable of storing a large number of objects. The first approach is to build an object data manager [CoM84,Dee86,Kim86,Lin86,Mae85,Ste86]. In this approach, the data manager stores objects that a program can fetch and store. The disadvantage of this approach is that a complete database management

† This research was supported by the National Science Foundation under Grant DCR-8507256 and by the Naval Electronics Systems Command Contract N0039-84-C-0089.

system (DBMS) must be written. A query optimizer is needed to support object queries (e.g., "fetch all *foo* objects where field *bar* is such-and-such"). Moreover, the optimizer must support the equivalent of relational joins because objects can include references to other objects. A transaction management system is needed to support shared access and to maintain data integrity should the software or hardware crash. Finally, protection and integrity systems are required to control access to objects and to maintain data consistency. These modules taken together account for a large fraction of the code in a DBMS. Proponents of this approach may argue that some of this functionality can be avoided. However, we believe that eventually all of this functionality will be required for the same reasons that it is included in conventional database management systems.

The second approach, and the one we are taking, is to store the object hierarchy in a relational database. The advantage of this approach is that we do not have to write a DBMS. A beneficial side-effect is that programs written in a conventional programming language can simultaneously access the data stored in the object hierarchy. The main objection to this approach has been that the performance of existing relational DBMS's has been inadequate. We believe this problem will be solved by using POSTGRES as the DBMS on which to implement the shared hierarchy. POSTGRES is a next-generation DBMS currently being implemented at the University of California, Berkeley [StR86]. It has a number of features, including procedure data types, precomputed procedures, and alerters, that can be used to implement the shared object hierarchy efficiently. A group at Intellicorp is also pursuing this approach [AbW86]. However, they do not store the complete object hierarchy in the database as we are planning to do.

Figure 1 shows the architecture of the proposed system. Each application process is connected to a database process that manages the shared database. The application program is presented a conventional view of the object hierarchy. As objects are referenced by the program, a run-time system retrieves them from the database. Objects retrieved from the database are stored in an object cache in the application process so that subsequent references to the object will not require another database retrieval. Object updates are propagated to the database and to other processes that have cached the object.

This paper describes how a shared object hierarchy can be implemented efficiently on POSTGRES. The POSTGRES mechanisms used to implement the shared object hierarchy are described in another paper in these proceedings [Sto86]. The remainder of the paper is organized as follows. Section 2

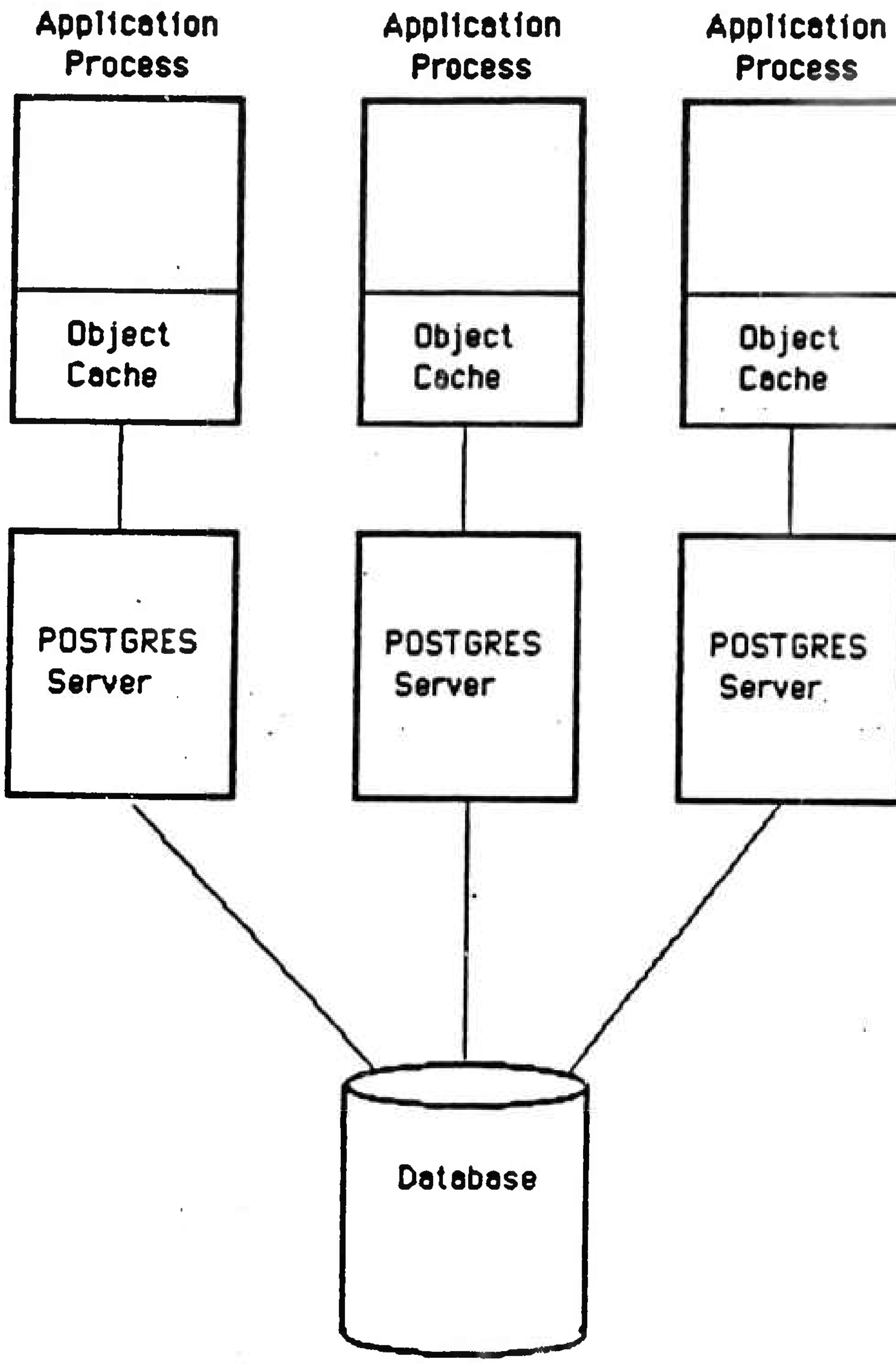

Figure 1. Process architecture.

describes the database representation for the object hierarchy. Section 3 describes the design of the object cache including strategies for improving the performance of fetching objects from the database. Section 4 discusses object updating and transactions. Section 5 describes the support for selecting and executing methods. And lastly, section 6 summarizes the paper.

2. Object Hierarchy Database Design

This section describes the object hierarchy model and explains how it is represented in a POSTGRES database.

Every object has a name, a type (called the object *class*), one or more parent classes (called *superclasses*), a set of local variables (called *instance variables*), and a set of procedures or methods that operate on the object (called *instance methods*). In an object-oriented system, the class of an object (i.e., the object type) is represented at run-time by an object. For example, figure 2 shows an example object hierarchy with two object classes: *Box* and *BorderBox*. The superclass of each class is indicated by a bold solid line with an arrow pointing to the parent class (e.g., *Box* is the superclass of *BorderBox*). By convention the class *Object* is at the top of the superclass hierarchy.

The example figure also shows class instances. There are two instances of *Box* (labeled *Box-1* and *Box-2*) and three instances of *BorderBox* (labeled *BorderBox-1*, *BorderBox-2*, and *BorderBox-3*). The class of each instance is indicated by a normal width line with an arrow that points at the class object for the instance. This relationship is also called the *instance-of* relationship. Since classes are represented by objects, they must also have a type or class (called the classes' *metaclass*). In most cases, the metaclass of a class object (e.g., *Box*) is an object named *Class*. For completeness and consistency, the *Class* object has a class named *MetaClass* which is its own class.

Since the object that represents a particular class is a normal object, it may have its own local variables. These variables, called *class variables*, are global to all instances of the class. They can be used to store data that is constant for all instances, that is computed from the instances (e.g., keeping a count of the number of instances), or that is used to pass data between instances.

Figure 3 shows object definitions for *Box* and *BorderBox*. Objects inherit the procedures and data defined in their superclasses. In this example, *BorderBox* inherits the instance variables *Origin*, *HorzLen*, and *VertLeng* and the class variable *NumberBoxes*. It also inherits the methods

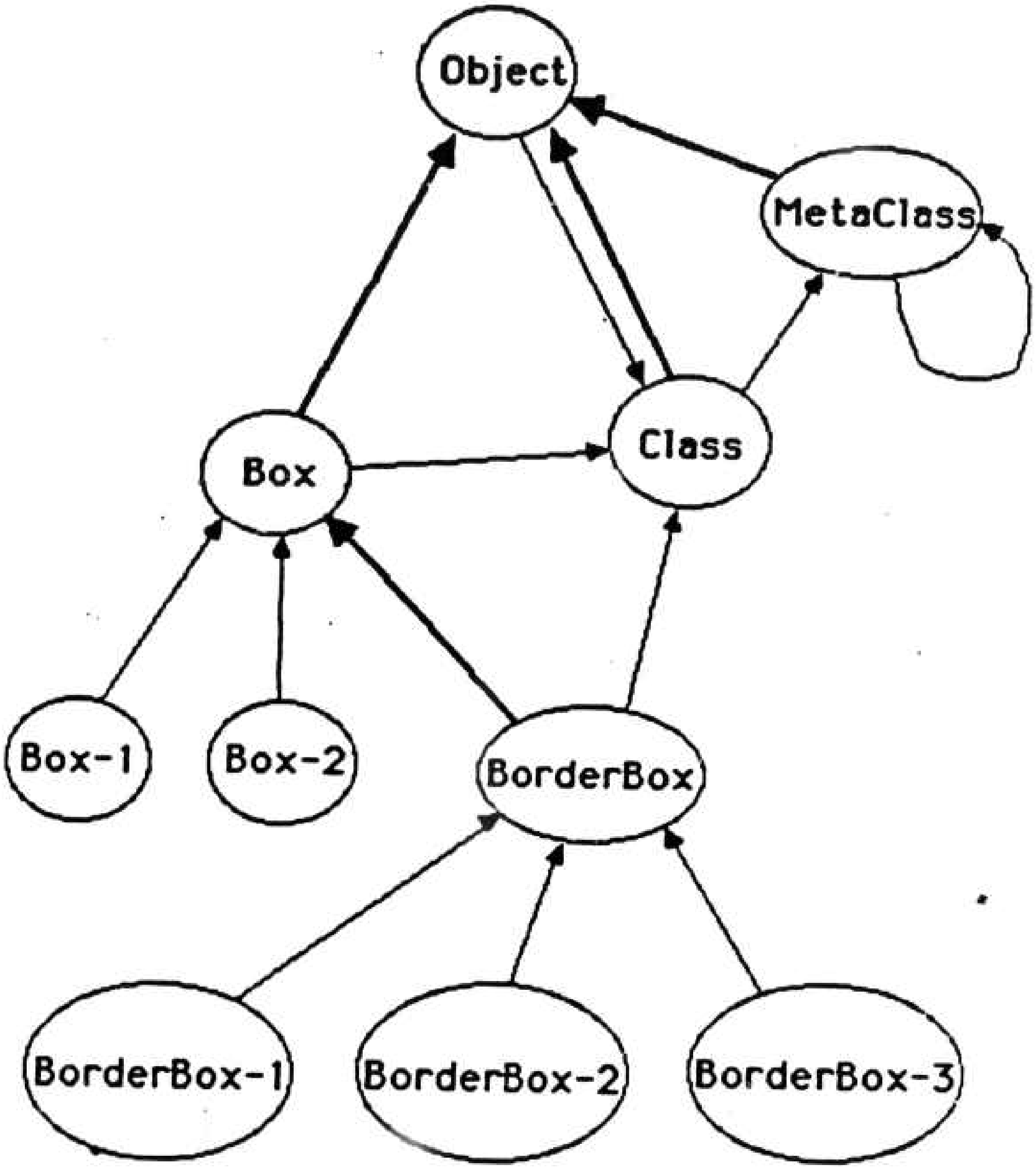

Figure 2. Example of an object hierarchy.

Box

MetaClass Class

Superclasses (Object)

Class Variables
```
        NumberBoxes                /* number of boxes created */
```

Instance Variables
```
        Origin                     /* origin of upper left corner */
        HorzLen                    /* horizontal length of box */
        VertLen                    /* vertical length of box */
```

Instance Methods
```
        Move(NewOrigin)            /* move box */
        Reshape(NewOrigin, NewHorzLen, NewVertLen)   /* reshape box */
        Shade(NewShade)            /* shade the interior of the box */
        Draw()                     /* draw the box */
```

3(a): *Box* object definition.

BorderBox

MetaClass Class

SuperClasses (Box)

Class Variables

Instance Variables
```
        BorderSize                 /* size of border */
```

Instance Methods
```
        SetBorder(NewBorderSize)        /* set border size */
        Reshape(NewOrigin, NewHorzLen, NewVertLen)   /* reshape box */
        EraseBorder()              /* erase the border */
        DrawBorder()               /* draw the border */
```

3(b): *BorderBox* object definition.

Figure 3. Example object definitions.

Move, *Shade*, and *Draw*. The method *Reshape* is not inherited from *Box* because a new definition for that method is given in *BorderBox*.

The database representation for this hierarchy requires three catalog relations (OBJECTS, SUPERS, and METHODS) and a relation for each object class. The *OBJECTS* relation keeps track of all objects:

OBJECTS(objid, name, owner, created,
        modified, instance-of, objrep)

where

objid	is a unique identifier for the object.
name	is the name of the object.
owner	is the object creator.
created	is the date/time when the object was created.
modified	is the date/time when the object was last modified.
instance-of	is the *objid* of the class object of this instance.
objrep	is a cached version of the memory representation of the object (described below).

The superclass relationship is represented in the *SUPERS* relation:

SUPERS(class, superclass, seqno)

where

class	is the *objid* of the class object.
superclass	is the *objid* of the parent class object.
seqno	is a sequence number that specifies the inheritance order in the case that a class has more than one superclass.

The superclass relationship is stored in a separate relation because an object can inherit variables and methods from more than one parent (i.e., multiple inheritance).

The sequence number in the SUPERS relation is needed to determine which variable or method should be inherited when more than one superclass has a variable or method with the same name. The inherited variable or method is determined by a precedence list of superclass objects. Different object models use difference precedence rules [StB86]. We plan to follow the CommonLoops model which defines a default rule for determining the precedence list, but allows the user to modify the rule [Boe85].

Methods are represented in the METHODS relation:

METHODS(objid, name, proc)

where

objid	is the identifier of the class object to which this method belongs.
name	is the name of the method.
proc	is the code for the method.

Methods are associated with class objects because they are the same for all objects in the class.

Instances of object classes are represented by tuples in a relation created for the class. The relation has an attribute for each instance variable. For example, the instance variables for the *Box* class are stored in the *BOX* relation:

BOX(objid, Origin, HorzLen, VertLen)

The creation of a *Box* instance causes a tuple to be appended to the *OBJECTS* and *BOX* relations.

Class variables are somewhat more difficult to represent. The straightforward approach would be to define a relation *CVARS* that contained a tuple for each class variable:

CVAR(objid, name, value)

where *objid* and *name* uniquely determine the class variable and *value* represents the current value of the variable. This solution requires a union type mechanism because values in different tuples have different types. POSTGRES does not support union types because they violate the relational tenet that all values in a column have the same type. One possible representation for class variables is to define a separate relation with a single tuple that holds the current values of the class variables. For example. the following relation would be defined for the *Box* class:

BOX-CVAR(objid, NumberBoxes)

This solution works but it introduces some representational overhead (the extra relation) and requires another join to fetch the definition of an object. Moreover, it does not take advantage of database system features that can be used to update the count automatically.

POSTGRES has a mechanism, namely, virtual fields, that may provide a better solution. Virtual fields are not stored with each tuple in a relation but appear to the user as if they were stored [Sto85]. The following command defines a virtual field that automatically recomputes the number of boxes for the *Box* class variable:

```
REPLACE DEMAND BOX(
        NumberBoxes = COUNT(BOX.boxid)
)
```

Any reference to *BOX.NumberBoxes* will execute the COUNT aggregate to determine the current number of boxes. Thus, the database maintains the correct count. The disadvantage of this representation is that the COUNT aggregate is executed every time the variable is referenced. A better approach is to define a registered procedure that contains a query to retrieve the count. The POSTGRES precomputable procedures mechanism can then be used to cache the answer to this query so that it does not have to be recomputed every time the virtual field is referenced.

Class variables that are not computable from the database can be represented by a virtual field that is assigned the current value as illustrated in the following command:

```
REPLACE DEMAND BOX(
        x = " ...current value... "
)
```

Given this definition, a reference to *BOX.x* in a query will return the current value of the class variable. The variable can be updated by redefining the virtual field. We plan to experiment with both the single tuple relation and virtual field approaches to determine which provides better performance.

This section described the object hierarchy model and a database design for storing it in a relational database. The next section describes the front-end process object cache and optimizations to improve the time required to fetch an object from the database.

3. Object Cache Design

The object cache must support three functions: object fetching, object updating, and method determination. This section describes the design for efficiently accessing objects. The next section describes the support for object updating and the section following that describes the support for method determination.

The major problem with implementing an object hierarchy on a relational database system is the time required to fetch an object. This problem arises because queries must be executed to fetch and update objects and because objects are decomposed and stored in several relations that must be joined to retrieve it from the database. Three strategies will be used to speed-up object fetch time: caching, precomputation, and prefetching. This section describes how these strategies will be implemented.

The application process will cache objects fetched from the database. The cache will be similar to a conventional Smalltalk run-time system [Kae81]. An object index will be maintained in main memory to allow the run-time system to determine quickly if a referenced object is in the cache. Each index entry will contain the *OBJECTS* relation tuple and the memory address of the object. All object references, even instance variables that reference other objects, will use the object identifier assigned to the object in the database (i.e., the *objid* attribute in the *OBJECTS* relation) to point to an object. These soft object pointers may slow the system down but they avoid the problem of mapping addresses when objects are moved between main memory and the database.[1] The object index will be hashed to speed-up object referencing.

Object caching can speed-up references to objects that have already been fetched from the database but it cannot speed-up the time required to fetch the object the first time it is referenced. The implementation strategy we will use to solve this problem is to precompute the memory representation of an object and to cache it in the *OBJECTS* relation. Suppose we are given the function *RepObject* that takes an object identifier and returns the memory representation of the object. A front-end process could execute *RepObject* and store the result back in the *OBJECTS* relation. This approach does not work because the precomputed representation must be invalidated if another process updates the object either through an operation on the object or an operation on the relation containing the object representation. For example, a user could run the following query to update all values of the variable *BorderSize* in instances of the object *BorderBox*

```
REPLACE BORDERBOX(
        BorderSize = new-value
)
```

[1] Most Smalltalk implementations use a similar scheme and it does not appear to be a bottleneck.

This update must invalidate all precomputed *BorderBox*'s.[2]

The other approach is to have the DBMS process execute *RepObject* and invalidate the cached result when necessary. POSTGRES supports precomputed procedure values that can be used to implement this approach. POSTQUEL commands and procedures can be stored as values of a relation attribute. A query that calls *RepObject* to compute the memory representation for the object is stored in the *objrep* attribute of the *OBJECTS* relation. For example, the following query is stored in *objrep*:

```
RETRIEVE (rep = RepObject($objid))
```

$objid refers to the object identifier in the tuple in which this query is stored (i.e., *OBJECTS.objid*). To retrieve the *OBJECTS* tuple and memory representation for the object named *foo*, the following query is executed:

```
RETRIEVE (o.objid, o.name, o.created, o.modified,
          o.instance-of, obj = o.objrep.rep)
FROM o IN OBJECTS
WHERE o.name = "foo"
```

The nested dot notation (*o.objrep.rep*) accesses values from the result tuples of the query stored in *objrep*. Executing the query causes *RepObject* to be called which returns the object memory representation.

This representation by itself does not alter the performance of fetching an object. The performance can be changed by instructing the DBMS to precompute the query in *objrep* (i.e., to cache the memory representation of the object in the *OBJECT* tuple). If this optimization is performed, fetching an object turns into a single relation, restriction query which can be carefully optimized to be very efficient. POSTGRES supports precomputation of values of type procedure or POSTQUEL. A POSTQUEL value marked as **precomputable** causes the database process to evaluate the command off-line and cache the result in the tuple.[3] Moreover, any database values retrieved by the POSTQUEL commands will be marked so that if they are updated, the cached result can be invalidated. This mechanism is described

[2] *BorderBox*'s cached in this process or other processes must also be invalidated. Object updating, cache consistency, and update propagation are discussed in the next section.

[3] The back-end checks that the command does not update the database and that any procedures called in the command also do not do updates so that precomputing the command will not introduce side-effect errors.

in greater detail elsewhere [StR86].

The **precomputable** attribute of a procedure or POSTQUEL value can be turned on or off for each object (i.e., it is an attribute of the value not the type). Consequently, the performance of the object run-time system can be tuned so that only frequently accessed objects are precomputed. And, because this attribute can be changed dynamically, precomputation of object representations can be modified at run-time depending on how the object is being used (e.g., is the object being edited or browsed).

The last implementation strategy to speed-up object referencing is prefetching. The basic idea is to fetch an object into the cache before it is referenced. The *HINTS* relation maintains a list of objects that should be prefetched when a particular object is fetched:

```
HINTS(fetch-object, hint-object)
```

When an object is fetched from the database, all *hint-object*'s for the object will be fetched at the same time. For example, after fetching an object, the following query can be run to prefetch some other objects:

```
RETRIEVE (o.objid, o.name, o.created, o.modified,
              o.instance-of, obj = o.objrep.rep)
FROM o IN OBJECTS, h IN HINTS
WHERE o.objid = h.hint-object
  AND h.fetch-object = fetched-object-identifier
```

If the prefetched objects have also been precomputed, this query will load a set of objects in one transaction.

We believe that with these three strategies object retrieval from the database can be implemented efficiently. Our attention thus far has been focussed on speeding up object fetching from the database. We will also have to manage the limited memory space in the object cache. An LRU replacement algorithm will be used to select infrequently accessed objects to be removed from the cache. We will also have to implement a mechanism to "pin down" objects that are not accessed frequently but which are critical to the execution of the system or are time consuming to retrieve.

This section described strategies for speeding up object fetching. The next section discusses object updating.

4. Object Updating and Transactions

This section describes the run-time support for updating objects. Two aspects of object updating are discussed: how the database representation of an object is updated (database concurrency and transaction management and how the update is propagated to other processes that have cached the object.

The run-time system in the front-end process specifies the desired update mode for an object when it is fetched from the database into the object cache. The system supports four update modes: local-copy, direct-update, deferred-update, and object-update. Local-copy mode makes a copy of the object in the cache. Updates to the object are not propagated to the database and updates by other processes are not propagated to the local copy. This mode is provided so that changes are valid only for the current session.

Direct-update mode treats the object as though it were actually in the database. Each update to the object is propagated immediately to the database. In other words, updating an instance variable in an object causes an update query to be run on the relation that represents instances of the object. A conventional database transaction model is used for these updates. Write locks are acquired when the update query is executed and they are released when it finishes (i.e., the update is a single statement transaction). Note that read locks are not acquired when an object is fetched into the cache. Updates to the object made by other processes are propagated to the cached object when the run-time system is notified that an update has occurred. The notification mechanism is described below. Direct-update mode is provided so that the application can view "live data."

Deferred-update mode saves object updates until the run-time system explicitly requests that they be propagated to the database. A conventional transaction model is used to specify the update boundaries. A begin transaction operation can be executed for a specific object. Subsequent variable accesses will set the appropriate read and write locks to ensure transaction atomicity and recoverability. The transaction is committed when an end transaction operation is executed on the object. Deferred-update mode is provided so that the application can make several updates atomic.

The last update mode supported by the system is object-update. This mode treats all accesses to the object as a single transaction. An intention-to-write lock is acquired on the object when it is first retrieved from the database. Other processes can read the object, but they cannot update it. Object updates are propagated to the database when the object is released from the cache. This mode is provided so that transactions can be expressed in terms of the object, not the database representation. However, note that this mode may reduce concurrency because the entire object is locked while it is in the object cache.

Thus far, we have only addressed the issue of propagating updates to the database. The remainder of this section will describe how updates are propagated to other processes that have cached the updated object. The basic idea is to propagate updates through the shared database. When a process retrieves an object, a database alerter [BuC79] is set on the object that will notify the process when it is updated by another process. When the alerter is trigger by another process, the process that set the alerter is notified. The value returned by the alerter to the process that set it is the updated value of the object. Note that the precomputed value of the object memory representation will be invalidated by the update so that it will have to be recomputed by the back-end. The advantage of this approach is that the process that updates an object does not have to know which processes want to be notified when a particular object is updated.

The disadvantages of this approach are that the database must be prepared to handle thousands of alerters and the time and resources required to propagate an update may be prohibitive. Thousands of alerters are required because each process will define an alerter for every object in its cache that uses direct-, deferred-, or object-update mode. An alerter is not required for local-copy mode because database updates by others are not propagated to the local copy. POSTGRES is being designed to support large databases of rules so this problem is being addressed.

The second disadvantage is the update propagation overhead. The remainder of this section describes two propagated update protocols, an alerter protocol and a distributed cache update protocol, and compares them. Figure 4 shows the process structure for the alerter approach. Each user process (UP) has a database process called its POSTGRES server (PS). The POSTMASTER process (PM) controls all POSTGRES servers. Suppose that UP_i updates an object in the database on which $K \leq N$ UP's have set an alerter. Figure 5 shows the protocol that is executed to propagate the updates to the other UP's. The cost of this propagated update is:

$2K+1$	process-to-process messages
1	database update
1	catalog query
1	object fetch

The object fetch is avoidable if the alerter returns the changed value. This optimization works for small objects but may not be reasonable for large objects.

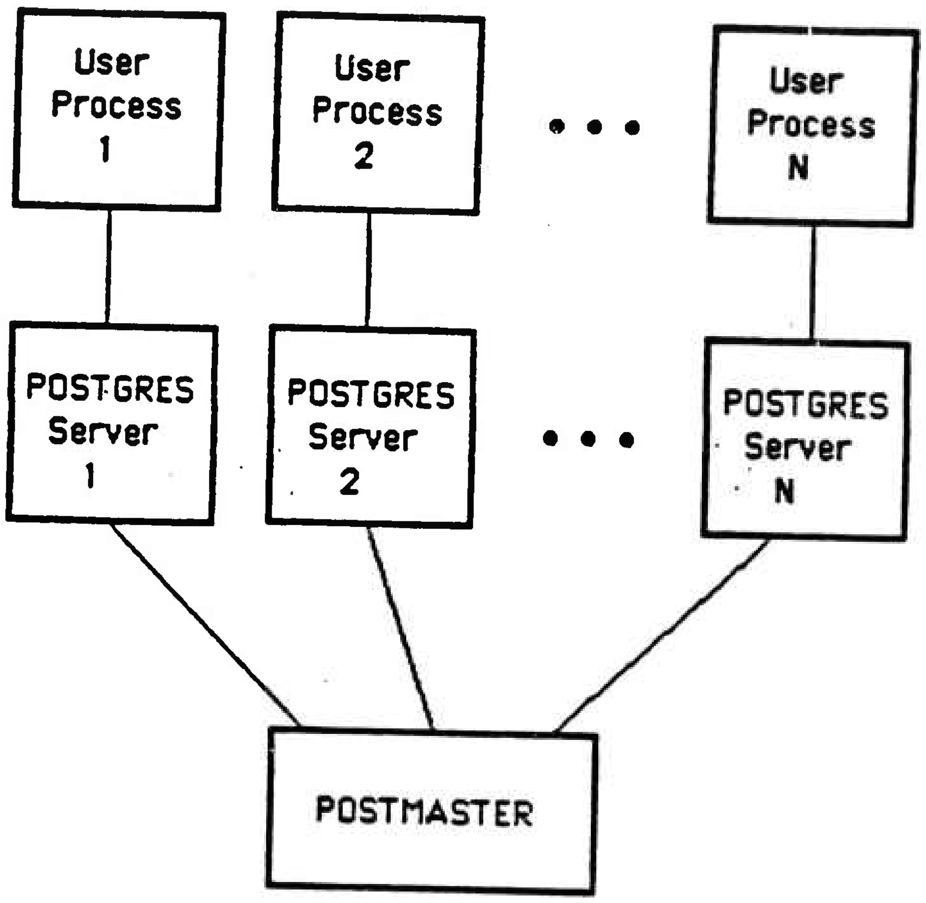

Figure 4. Process structure for the alerter approach.

1. UP_i updates the database.
2. PS_i sends a message to PM indicating which alerters were tripped.
3. PM queries the alerter catalog to determine which PS's set the alerters.
4. PM sends a message to PS_j for each alerter.
5. Each PS_j sends a message to UP_j indicating that the alerter has been tripped.
6. Each PS_j refetches the object.

Figure 5. Propagated update protocol for the alerter approach.

The alternative approach to propagate updates is to have the user processes signal each other that an update has occurred. We call this approach the *distributed cache update* approach. The process structure is similar to that shown in figure 4, except that each UP must be able to broadcast a message to all other UP's. Figure 6 shows the distributed cache update protocol. This protocol uses a primary site update protocol. If UP_i does not have the update token signifying that it is the primary site for the object, it sends a broadcast message to all UP's requesting the token. The UP that has the token sends it to UP_i. Assuming that UP_i does not have the update token, the cost of this protocol is:

2	broadcast messages
1	process-to-process message
1	database update
1	object fetch

One broadcast message and the process-to-process message are eliminated if UP_i already has the update token. The advantage of this protocol is that a multicast protocol can be used to implement the broadcast messages in a way that is more efficient than sending N process-to-process messages. Of course, the disadvantage is that UP's have to examine all update signals to determine whether the updated object is in its cache.

1. UP_i acquires the update token for the object.
2. UP_i updates the database.
3. UP_i broadcasts to all UP's that the object has been updated.
4. Each UP_j that has the object in its cache refetches it.

Figure 6. Propagated update protocol for the distributed cache approach.

Assume that the database update and object fetch take the same resources in both approaches and that the alerter catalog is cached in main memory so the catalog query does not have to read the disk in the alerter approach. With these assumptions, the comparison of these two approaches comes down to the cost of 2 broadcast messages versus 2K process-to-process messages. If objects are cached in relatively few UP's (i.e., K << N) and broadcast messages are efficient, the distributed cache update appears better. On the other hand, if K is larger, so the probability of doing 2 broadcasts goes up, and broadcasts are inefficient, the alerter approach appears better. We have chosen the alerter approach because an efficient multicast protocol does not exist but the alerter mechanism will exist in POSTGRES. If this approach is too slow, we will have to tune the alerter code or implement the multicast protocol.

This section described the mechanisms for updating shared objects. The last operation that the run-time system must support is method determination which is discussed in the next section.

5. Method Determination

Method determination is the action taken to select the method to be executed when a message is sent to a particular object. The message contains the name of the method to be executed, called the *method selector*. The algorithm for selecting the method is to look for a method with the same selector name in the object to which the message was sent. If the method is not defined locally, the method is looked for in the superclasses of the object. This process continues until either the method is found or the *Object* object is reached which indicates the method does not exist.

Conventional object-oriented systems implement a cache of recently called methods to speed-up method determination [GoR83]. The cache is typically a hash table that maps an object identifier of the receiving object and a method selector to the entry address of the method to be executed. If the desired object-selector pair is not in the table, the standard look-up algorithm is invoked. In memory resident Smalltalk systems, this strategy has proven to be very good because high hit ratios have been achieved with modest cache sizes (e.g., 95% with 2K entries) [Kra83].

We will adapt the method cache idea to the database environment. A method index relation will be computed that indicates which method should be called for each object-selector pair. The data will be stored in the *DM* relation defined as follows:

DM(use-obj, method-name, def-obj)

where

use-obj	is the receiving object.
method-name	is the name of the method.
def-obj	is the object in which the method is defined.

Given this relation, the method to be executed can be retrieved from the database by the following query:

```
RETRIEVE (m.proc)
FROM m IN METHODS, d IN DM
WHERE m.objid = d.def-obj
  AND d.use-obj = "...receiving object..."
  AND d.method-name = "...message selector..."
```

As mentioned in section 2, both the source and binary representations of a procedure are stored in the *proc* attribute in the *METHODS* relation.

Method code will be cached in the front-end process so that the database will not have to be queried for every procedure call. Procedures in the cache will have to be invalidated if another process modifies the method definition or the inheritance hierarchy. Database alerters will be used to signal object changes that require invalidating cache entries. We will also support a check-in/check-out protocol for objects so that production programs can isolate their object hierarchy from changes being made by others doing development [Kat83].

The DM relation can be precomputed for all objects in the hierarchy and incrementally updated as the hierarchy is modified. Depending on the rules used for multiple inheritance, the computation of DM could be a single query or a more complicated sequence of queries.

This section described a shared index that can be used for method determination.

6. Summary

This paper described a proposed implementation of a shared object hierarchy in a POSTGRES database. Objects accessed by an application program are cached in the application process. Precomputation and prefetching are used to reduce the time to retrieve objects from the database. Several update modes were defined that can be used to control the concurrency allowed. Database alerters are used to propagate updates to copies of objects in other caches. A number of features in POSTGRES will be exploited to implement the system, including: procedure and POSTQUEL data types, precomputed procedures and queries, database alerters, and virtual fields.

References

[AbW86] R. M. Abarbanel and M. D. Williams, A Relational Representation for Knowledge Bases, Unpublished manuscript, Apr. 1986.

[Boe85] D. B. Bobrow and et.al., "COMMONLOOPS: Merging Common Lisp and Object-Oriented Programming", Intelligent Systems Laboratory ISL-85-8, Xerox PARC, Aug. 1985.

[BuC79] O. P. Buneman and E. K. Clemons, "Efficiently Monitoring Relational Databases", *ACM Trans. Database Systems* , Sep. 1979, 368-382.

[CoM84] G. Copeland and D. Maier, "Making Smalltalk a Database System", *Proc. 1984 ACM-SIGMOD Int. Conf. on the Mgt. of Data*, June 1984.

[Dee86] N. P. Derrett and et.al., "An Object-Oriented Approach to Data Management", *Proc. 1986 IEEE Spring Compcon*, 1986.

[GoR83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison Wesley, Reading, MA, May 1983.

[Kae81] T. Kaehler, "Virtual Memory for an Object-Oriented Language", *Byte 6*, 8 (Aug. 1981).

[KaK83] T. Kaehler and G. Krasner, "LOOM – Large Object-Oriented Memory for Smalltalk-80 Systems", in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner (editor), Addison Wesley, Reading, MA, May 1983.

[Kat83] R. Katz, "Managing the Chip Design Database", *Computer Magazine 16*, 12 (Dec. 1983).

[Kim86] W. Kim, Personal communication., 1986.

[Kra83] G. Krasner, ed., *Smalltalk-80: Bits of History, Words of Advice*, Addison Wesley, Reading, MA, May 1983.

[Lin86] M. Linton, Personal communication., 1986.

[Mae85] D. Maier and et.al., Development of an Object-Oriented DBMS. Unpublished manuscript, June 1985.

[Ste86] M. Stefik, Personal communication., 1986.

[StB86] M. Stefik and D. G. Bobrow, "Object-Oriented Programming: Themes and Variations", *The AI Magazine 6*, 4 (Winter 1986), 40-62.

[Sto85] M. Stonebraker, "Triggers and Inference in Data Base Systems", *Proc. Islamoora Conference on Expert Data Bases*, Feb. 1985. To appear as a Springer-Verlag book.

[Sto86] M. R. Stonebraker, Object Management in POSTGRES Using Procedures, Submitted to the OODBS Workshop, June 1986.

[StR86] M. R. Stonebraker and L. A. Rowe, "The Design of POSTGRES", *Proc. 1986 ACM-SIGMOD Int. Conf. on the Mgt. of Data*, June 1986.